CSE 464: Software Quality Assurance and Testing
Course Project Part-1

Vajravel Conjeevaram Manigandan
1225428152
vconjeev@asu.edu

Introduction:

This is a Java based application that can parse graphs in DOT format, manipulate graphs (e.g., add nodes, add edges), and output graphs in DOT format and graphics (e.g., jpg and png).

This project was built using **IntelliJ** as the IDE, **Java 21** as the Java Development Kit (JDK), **Maven** to manage dependencies, and the **graphviz-java** was used to parse the DOT file into objects in java and vice versa. Classes such as Graph Manager, Graph, Node, and Edge were implemented to perform operations such as coding the features, creating a graph, adding nodes or edges on the given graph.

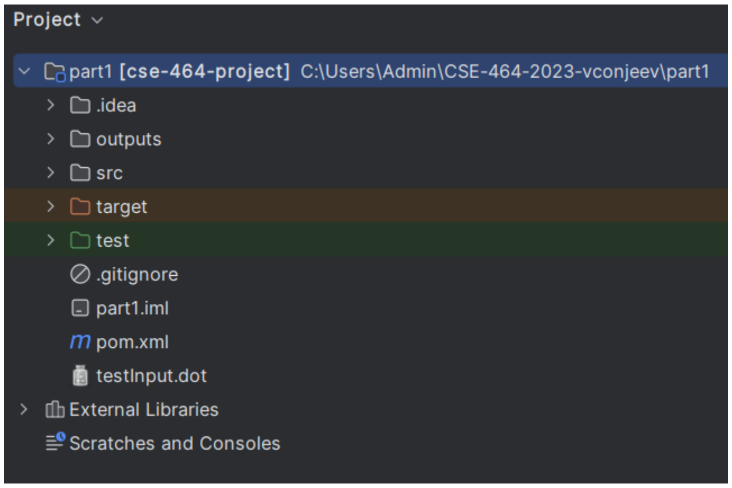
NOTE: ***If there are any dependency issues, please resolve them using mvn install in the command line terminal or resolve them using maven in IntelliJ.***

Access the project:

Every step of the project has been tracked using Github, and can be accessed via the link: https://github.com/v-Cm/CSE-464-2023-vconjeev

Project Structure:

The structure of the project is shown below.



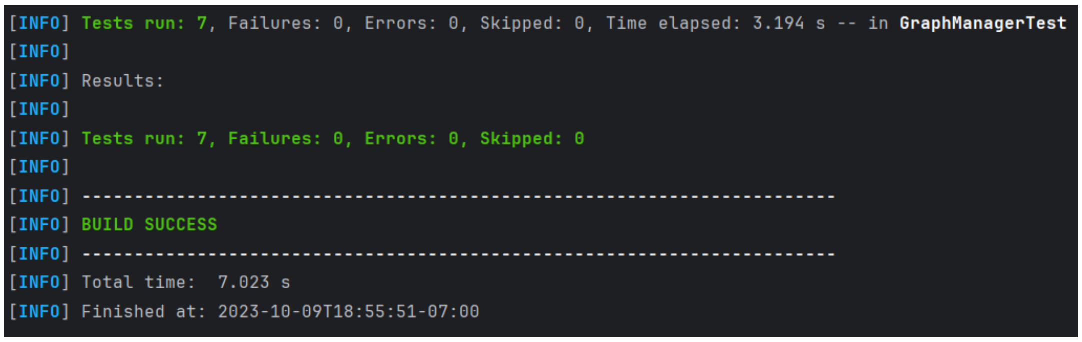
```
Project
part1 [cse-464-project]  C:\Users\Admin\CSE-464-2023-vconjeev\part1
  .idea
  outputs
  src
  target
  test
  .gitignore
  part1.iml
  pom.xml
  testInput.dot
External Libraries
Scratches and Consoles
```

Run the project:

Clone or download the project folder to your system. If you are using an IDE (IntelliJ) to access the project, you can simply run the **Application.java** file in the **/part1/src/main/java/** path. Or, if you are using the CLI/terminal you can type **mvn package** to compile and run the project.

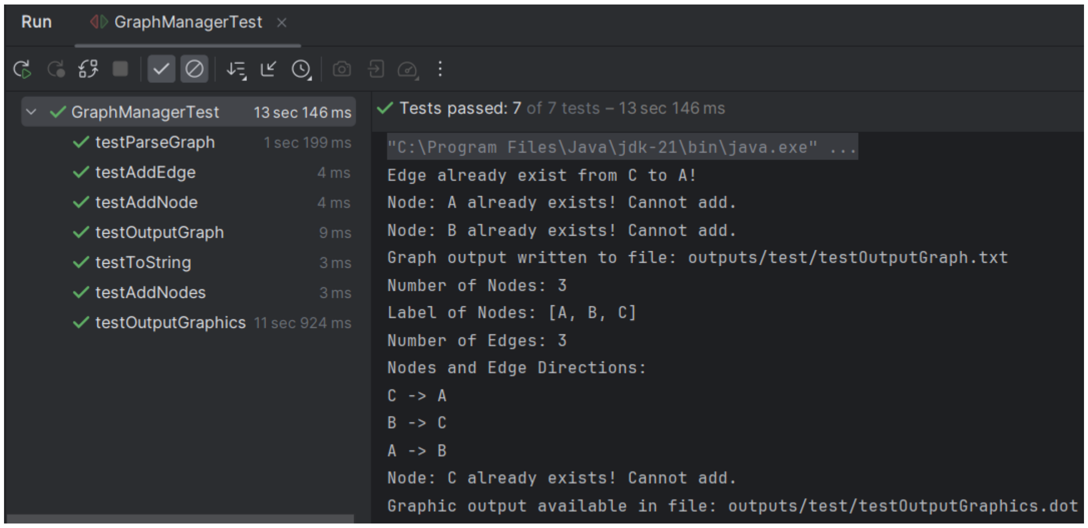
Test the project:

The application has several features in it (which will be described in detail in the following section) and the expected outputs for each features have been provided either in the **GraphManagerTest.java** file in the /part1/test/ path, or as files in the /**part1/outputs/expected** directory. Again, you can run **mvn test** or **mvn package** on the terminal,



```
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 3.194 s -- in GraphManagerTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  7.023 s
[INFO] Finished at: 2023-10-09T18:55:51-07:00
```

or run the testcases in the **GraphManagerTest.java** file manually.



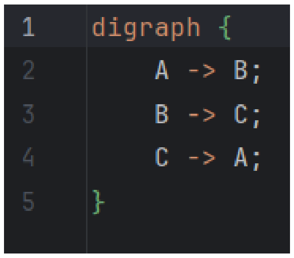
```
GraphManagerTest      13 sec 146 ms
  testParseGraph       1 sec 199 ms
  testAddEdge                  4 ms
  testAddNode                  4 ms
  testOutputGraph              9 ms
  testToString                 3 ms
  testAddNodes                 3 ms
  testOutputGraphics  11 sec 924 ms
```

```
Tests passed: 7 of 7 tests – 13 sec 146 ms
"C:\Program Files\Java\jdk-21\bin\java.exe" ...
Edge already exist from C to A!
Node: A already exists! Cannot add.
Node: B already exists! Cannot add.
Graph output written to file: outputs/test/testOutputGraph.txt
Number of Nodes: 3
Label of Nodes: [A, B, C]
Number of Edges: 3
Nodes and Edge Directions:
C -> A
B -> C
A -> B
Node: C already exists! Cannot add.
Graphic output available in file: outputs/test/testOutputGraphics.dot
```

About the features:

All the features required in part 1 are added into a Class named “GraphManager.java” which is in the folder: part1/src/main/java.

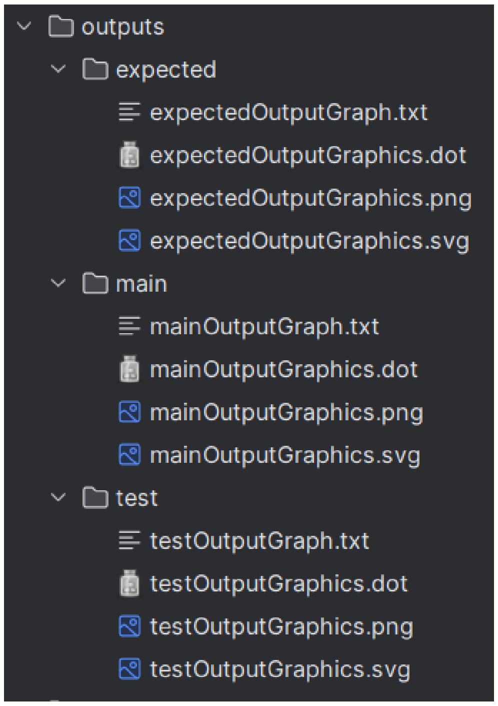
Input: The input DOT file is **testInput.dot** (which can be edited however you want).



```
digraph {
        A -> B;
        B -> C;
        C -> A;
}
```

testInput.dot

Output: All the outputs are generated in the outputs directory as detailed below:



```
outputs
    expected
        expectedOutputGraph.txt
        expectedOutputGraphics.dot
        expectedOutputGraphics.png
        expectedOutputGraphics.svg
    main
        mainOutputGraph.txt
        mainOutputGraphics.dot
        mainOutputGraphics.png
        mainOutputGraphics.svg
    test
        testOutputGraph.txt
        testOutputGraphics.dot
        testOutputGraphics.png
        testOutputGraphics.svg
```

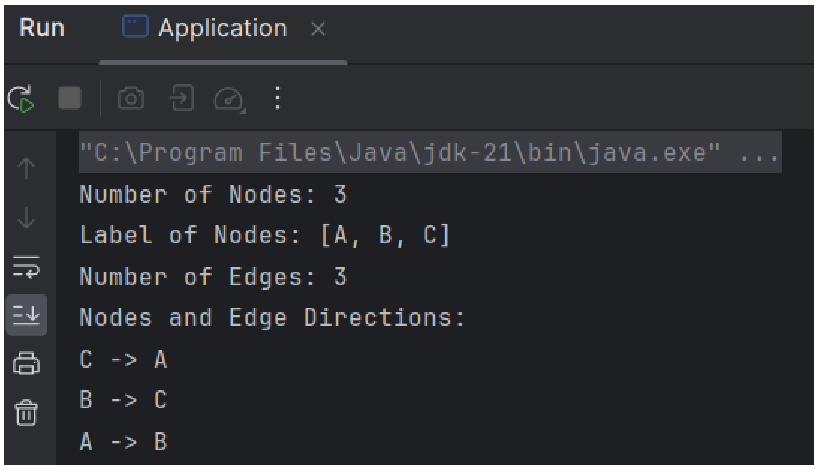
Outputs directory

- expected - contains the expected output files
- main - contains the outputs generated by running the main function in Application.java
- test - contains the outputs generated by the testcases in GraphManager.java

Feature 1: Parse a DOT graph file to create a graph.
Github commit link.

This feature implements three APIs:

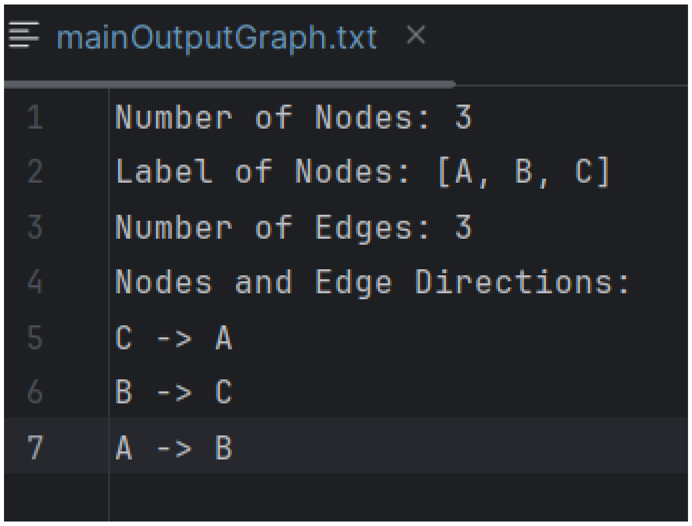
- The parseGraph(String filepath) API creates a directed graph object based on the description in the testInput.dot file.
- The toString() API prints the graph to the console.



```
"C:\Program Files\Java\jdk-21\bin\java.exe" ...
Number of Nodes: 3
Label of Nodes: [A, B, C]
Number of Edges: 3
Nodes and Edge Directions:
C -> A
B -> C
A -> B
```

Expected output for toString()

- The outputGraph(String filepath) API copies the above API's result to a text file.



```
Number of Nodes: 3
Label of Nodes: [A, B, C]
Number of Edges: 3
Nodes and Edge Directions:
C -> A
B -> C
A -> B
```

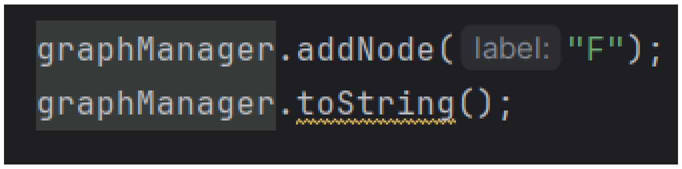
Output as shown in the mainOutputGraph.txt file

Feature 2: Adding nodes from the imported graph.

Github commit link.

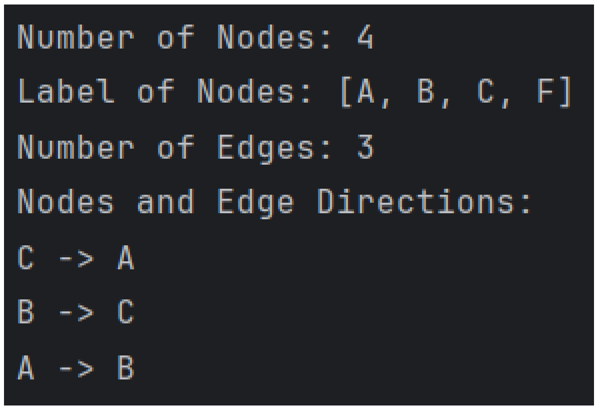
This feature implements two APIs:

- The addNode(String label) API adds a node and checks for duplicates.



```
graphManager.addNode( label: "F");
graphManager.toString();
```

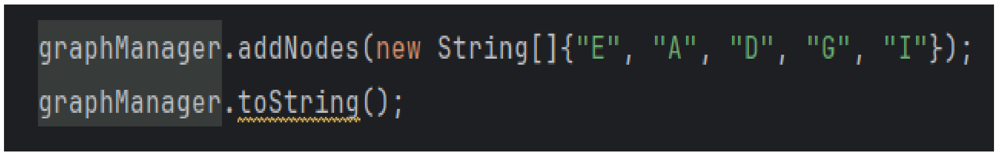
Invoking the addNode API



```
Number of Nodes: 4
Label of Nodes: [A, B, C, F]
Number of Edges: 3
Nodes and Edge Directions:
C -> A
B -> C
A -> B
```

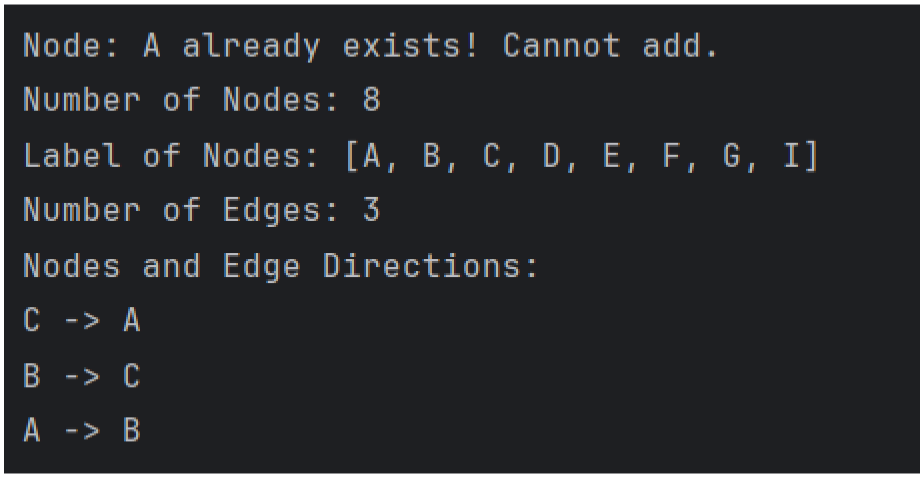
displaying the changes reflected by calling toString()

- The addNodes(String[] label). API adds a list of nodes and checks for duplicates as well.



```
graphManager.addNodes(new String[]{"E", "A", "D", "G", "I"});
graphManager.toString();
```

Invoking the addNodes API



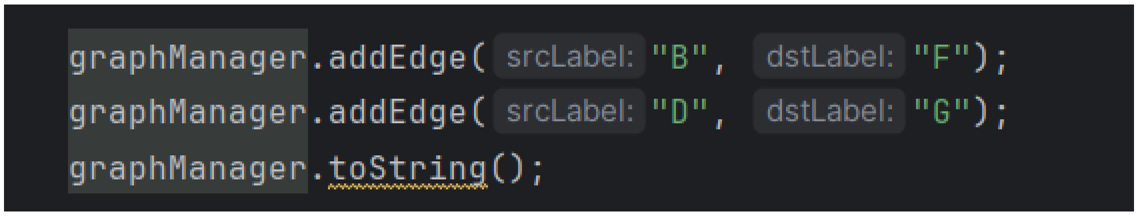
```
Node: A already exists! Cannot add.
Number of Nodes: 8
Label of Nodes: [A, B, C, D, E, F, G, I]
Number of Edges: 3
Nodes and Edge Directions:
C -> A
B -> C
A -> B
```

displaying the changes reflected by calling toString()

Feature 3: Adding edges from the imported graph.

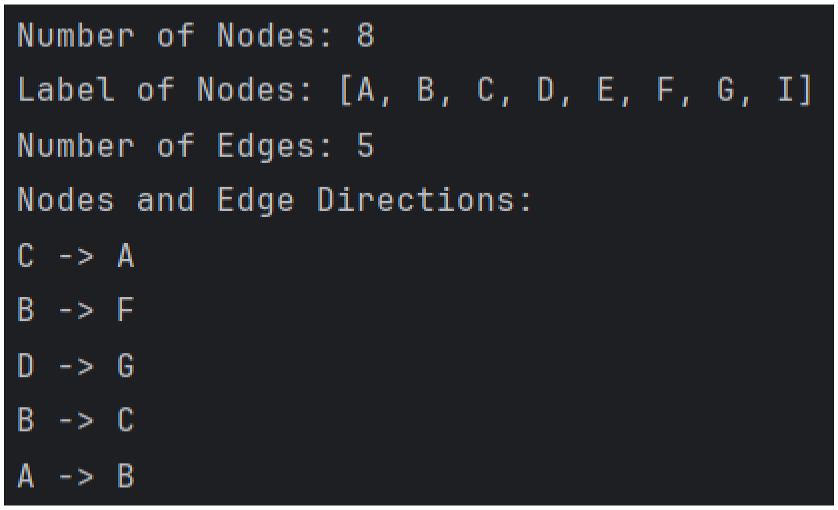
Github commit link.
This feature implements one API:

- The addEdge(String srcLabel, String dstLabel) API adds an edge and checks for duplicates.



```
graphManager.addEdge( srcLabel: "B",  dstLabel: "F");
graphManager.addEdge( srcLabel: "D",  dstLabel: "G");
graphManager.toString();
```

Invoking the addEdge API

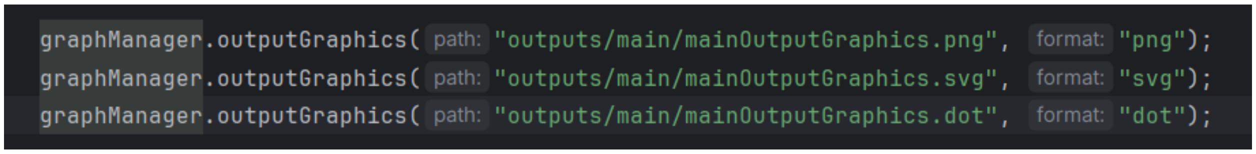


```
Number of Nodes: 8
Label of Nodes: [A, B, C, D, E, F, G, I]
Number of Edges: 5
Nodes and Edge Directions:
C -> A
B -> F
D -> G
B -> C
A -> B
```

displaying the changes reflected by calling toString()

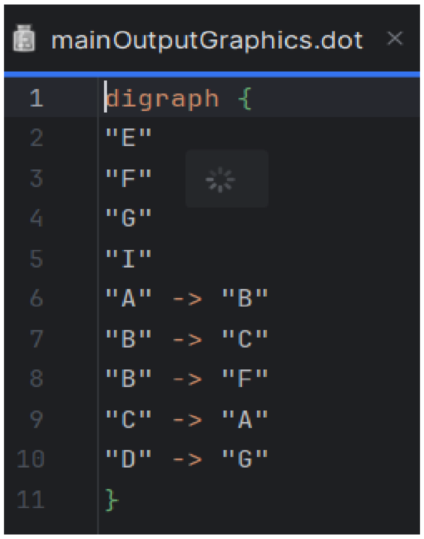
Feature 4: Output the imported graph into a DOT file or graphics.
Github commit link.
This feature implements one API:

- The outputGraphics(String path, String format) API converts the graph to a DOT, PNG, or SVG file and returns it to the path requested.



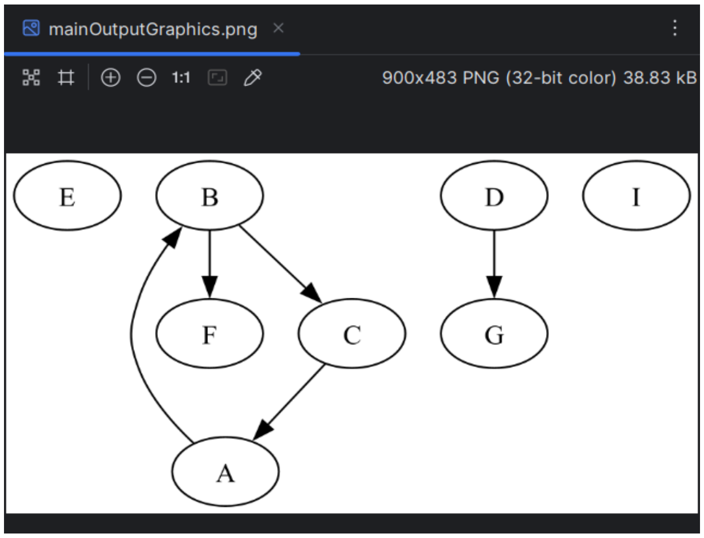
```
graphManager.outputGraphics( path: "outputs/main/mainOutputGraphics.png",  format: "png");
graphManager.outputGraphics( path: "outputs/main/mainOutputGraphics.svg",  format: "svg");
graphManager.outputGraphics( path: "outputs/main/mainOutputGraphics.dot",  format: "dot");
```

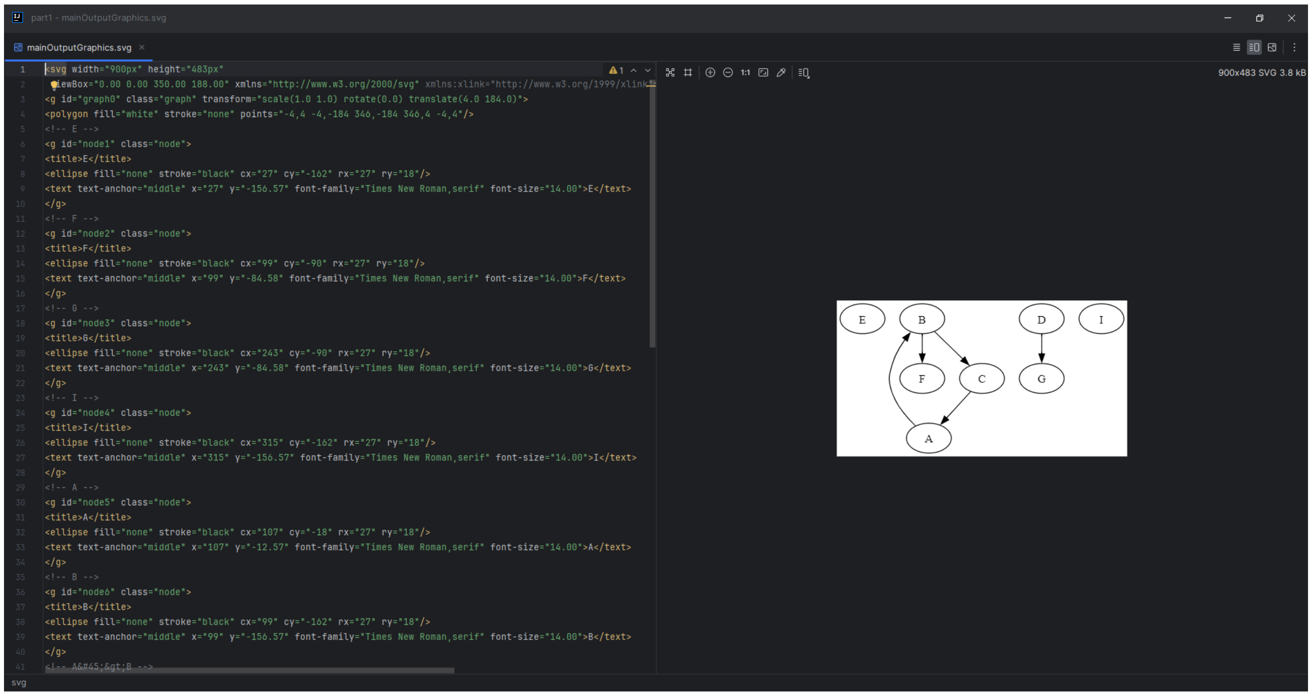
Invoking the outputGraphics API with all the supported formats



mainOutputGraphics.dot

```
digraph {
"E"
"F"
"G"
"I"
"A" -> "B"
"B" -> "C"
"B" -> "F"
"C" -> "A"
"D" -> "G"
}
```





part1 - mainOutputGraphics.svg

mainOutputGraphics.svg

```
<svg width="900px" height="483px"
 iewBox="0.00 0.00 350.00 188.00" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink
<g id="graph0" class="graph" transform="scale(1.0 1.0) rotate(0.0) translate(4.0 184.0)">
<polygon fill="white" stroke="none" points="-4,4 -4,-184 346,-184 346,4 -4,4"/>
<!-- E -->
<g id="node1" class="node">
<title>E</title>
<ellipse fill="none" stroke="black" cx="27" cy="-162" rx="27" ry="18"/>
<text text-anchor="middle" x="27" y="-156.57" font-family="Times New Roman,serif" font-size="14.00">E</text>
</g>
<!-- F -->
<g id="node2" class="node">
<title>F</title>
<ellipse fill="none" stroke="black" cx="99" cy="-90" rx="27" ry="18"/>
<text text-anchor="middle" x="99" y="-84.58" font-family="Times New Roman,serif" font-size="14.00">F</text>
</g>
<!-- G -->
<g id="node3" class="node">
<title>G</title>
<ellipse fill="none" stroke="black" cx="243" cy="-90" rx="27" ry="18"/>
<text text-anchor="middle" x="243" y="-84.58" font-family="Times New Roman,serif" font-size="14.00">G</text>
</g>
<!-- I -->
<g id="node4" class="node">
<title>I</title>
<ellipse fill="none" stroke="black" cx="315" cy="-162" rx="27" ry="18"/>
<text text-anchor="middle" x="315" y="-156.57" font-family="Times New Roman,serif" font-size="14.00">I</text>
</g>
<!-- A -->
<g id="node5" class="node">
<title>A</title>
<ellipse fill="none" stroke="black" cx="107" cy="-18" rx="27" ry="18"/>
<text text-anchor="middle" x="107" y="-12.57" font-family="Times New Roman,serif" font-size="14.00">A</text>
</g>
<!-- B -->
<g id="node6" class="node">
<title>B</title>
<ellipse fill="none" stroke="black" cx="99" cy="-162" rx="27" ry="18"/>
<text text-anchor="middle" x="99" y="-156.57" font-family="Times New Roman,serif" font-size="14.00">B</text>
</g>
<!-- A&#45;&gt;B -->
```

Displaying the outputs in .dot, .png, and .svg respectively. These files will be available in the outputs directory

NOTE: You don’t have to call toString() repeatedly as I did above. I did so to show the changes made to the input graph after calling each API from every feature.